

Parallel Programming in Python

Süleyman Savaş

Introduction and Motivation

- Today's applications have high performance demands and sometimes a single core is not enough
- To achieve higher performances, the applications can be divided into several parts which can run in parallel on different cores or processors

Thread-based Parallelism

- Running several threads is similar to running several different programs concurrently
- Threads run in the same process and share the memory
- Not much memory requirement
- Preemptive
- Can sleep (yielding)
- *Threading* Module in Python



Creating Thread using *Threading* Module

- Define a new subclass of the *Thread* class
- Override the *__init__(self [,args])* method to add additional arguments
- Override the *run(self [,args])* method to implement what the thread should do when started
- Create instance of subclass
- Call the *start()* method

Process-based Parallelism

- Similar to threading
- Processes run independently
- Slightly more memory requirement
- More robust
- Slightly more overhead
- ***Multiprocessing*** module in Python



Multiprocessing Module

- Similar to *Threading* module

Supports:

- Creating a process
 - Create a *Process* object and call the *start()* method
- Communication
 - Exchanging objects
 - Queues
 - Pipes
 - Sharing states
 - Shared memory
 - Server process
- Synchronization

Queue & Pipe

- Queue is a buffer which is shared by processes
- Pipe is a bi-directional connection channel between two processes

from multiprocessing import Process, Queue

```
def f(q):  
    q.put([42, None, 'hello'])  
  
if __name__ == '__main__':  
    q = Queue()  
    p = Process(target=f, args=(q,))  
    p.start()  
    print(q.get()) # prints "[42, None, 'hello']"  
    p.join()
```

from multiprocessing import Process, Pipe

```
def f(conn):  
    conn.send([42, None, 'hello'])  
    conn.close()  
  
if __name__ == '__main__':  
    parent_conn, child_conn = Pipe()  
    p = Process(target=f, args=(child_conn,))  
    p.start()  
    print(parent_conn.recv()) # prints "[42, None,  
'hello']"  
    p.join()
```



Shared Memory & Server Process

- Data can be stored in a shared memory map using *Value* or *Array*

```
from multiprocessing import Process, Value, Array
```

```
def f(n, a):  
    n.value = 3.1415927  
    for i in range(len(a)):  
        a[i] = -a[i]
```

```
if __name__ == '__main__':  
    num = Value('d', 0.0)  
    arr = Array('i', range(10))
```

```
    p = Process(target=f, args=(num, arr))  
    p.start()  
    p.join()
```

```
    print(num.value)  
    print(arr[:])
```

- A manager object returned by *Manager()* controls a server process which holds Python objects and allows other processes to manipulate them
- More flexible but slower

```
from multiprocessing import Process, Manager
```

```
def f(d, l):  
    d[1] = '1'  
    d['2'] = 2  
    l.reverse()
```

```
if __name__ == '__main__':  
    with Manager() as manager:  
        d = manager.dict()  
        l = manager.list(range(10))
```

```
    p = Process(target=f, args=(d, l))  
    p.start()  
    p.join()
```

```
    print(d)  
    print(l)
```


Synchronization

- *multiprocessing* module contains equivalents of all the synchronization primitives from the *threading* module

```
from multiprocessing import Process, Lock
```

```
def f(l, i):
```

```
    l.acquire()
```

```
    print('hello world', i)
```

```
    l.release()
```

```
if __name__ == '__main__':
```

```
    lock = Lock()
```

```
    for num in range(10):
```

```
        Process(target=f, args=(lock, num)).start()
```

- Without using the lock, output from the different processes are liable to get all mixed up.



Parallel Python (pp) Module

- Provides mechanism for parallel execution of python code on **SMP** (systems with multiple processors or cores) and **clusters**
- Automatic detection of the optimal configuration
- Dynamic processors allocation & Dynamic load balancing
- Low overhead for subsequent jobs with the same function (transparent caching is implemented to decrease the overhead)
- Fault-tolerance (if one of the nodes fails tasks are rescheduled on others)

Parallel Python (pp) Module

- Define the method(s) to be run on the processors/cores/cluster members
- Tuple the servers to connect with
- Create a job server with a number of workers
- Submit job(s) to the server
- The jobs will be distributed to the workers automatically
- Optionally: read the results

PyCUDA

- Gives Pythonic access to Nvidia's CUDA parallel computation API.
- Base layer is written in C++
- All CUDA errors are automatically translated into Python exceptions
- Includes code for interoperability with OpenGL
- Automatically infers what cleanup is necessary and does it

PyCUDA Example

```
import pycuda.autoint
import pycuda.driver as drv
import numpy

from pycuda.compiler import SourceModule
mod = SourceModule("""
__global__ void multiply_them(float *dest, float *a, float *b)
{
    const int i = threadIdx.x;
    dest[i] = a[i] * b[i];
}
""")

multiply_them = mod.get_function("multiply_them")

a = numpy.random.randn(400).astype(numpy.float32)
b = numpy.random.randn(400).astype(numpy.float32)

dest = numpy.zeros_like(a)
multiply_them(
    drv.Out(dest), drv.In(a), drv.In(b),
    block=(400,1,1), grid=(1,1))

print dest-a*b
```

- On the surface, this program will print a screenful of zeros. Behind the scenes, a lot more interesting stuff is going on:
- PyCUDA has compiled the CUDA source code and uploaded it to the card.
- PyCUDA's *numpy* interaction code has automatically allocated space on the device, copied the *numpy* arrays *a* and *b* over, launched a 400x1x1 single-block grid, and copied *dest* back.
- The data can be kept on the card between kernel invocations—no need to copy data all the time.
- No cleanup code is needed. PyCUDA will automatically infer what cleanup is necessary and do it.

Other modules/libs

- **dispy** - Python Framework for Distributed and Parallel Computing
- **delegate** - fork-based process creation with pickled data sent through pipes
- **forkmap** (original) - fork-based process creation using a function resembling Python's built-in map function (*Unix, Mac, Cygwin*).
- **forkfun** (modified) - fork-based process creation using a function resembling Python's built-in map function (*Unix, Mac, Cygwin*). (New version from July-2011 with modifications)
- **ppmap** - variant of forkmap using pp(parallel python) to manage the subprocesses (*Unix, Mac, Cygwin*)
- **POSH - Python Object Sharing** is an extension module to Python that allows objects to be placed in shared memory. POSH allows concurrent processes to communicate simply by assigning objects to shared container objects. (*POSIX/UNIX/Linux only*)
- **pprocess** (previously parallel/pprocess) - fork-based process creation with asynchronous channel-based communications employing pickled data (*currently only POSIX/UNIX/Linux, perhaps Cygwin*)
- **PyCSP - Communicating Sequential Processes** for Python allows easy construction of processes and synchronised communication.
- **remoteD** - fork-based process creation with a dictionary-based communications paradigm (*platform independent, according to PyPI entry*)
- **Pydra** - A Distributed Computing Framework For Python

References

- <http://wiki.python.org/moin/ParallelProcessing>
- documen.tician.de/pycuda/
- <https://pypi.python.org/pypi/pycuda>
- <http://www.praetorian.com/blog/multi-core-and-distributed-programming-in-python>
- <http://www.parallelpython.com/>
- http://www.tutorialspoint.com/python/python_multithreading.htm
- <http://docs.python.org>